Module 10
Arrays and Grids

Tamer Abdou, PhD

IND 830 - Python Programming for Data Science

Data Science Laboratory

Ryerson University

Ryerson University
The Chang School of Continuing Education

Outline

1 Introduction to Collections

2 Implementation of Collections

Ryerson University The Chang School of Continuing Education

Collections

1 *Introduction to Collections*
Categories of Collections
Operations on Collections

2 *Implementation of Collections*

Categories of Collections

- Linear Collections
- Hierarchical Collections
- Graph Collections
- Unordered Collections

Linear Collections

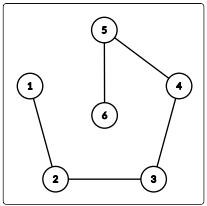


Figure 10.1: Linear Collection

- Ordered by position
- Each node except the first has a unique predecessor
- Each node except the last has a unique successor
- Node 2's predecessor is Node 1, and Node 2's successor is Node 3
- Example: A line of customers waiting at a bank

Hierarchical Collections

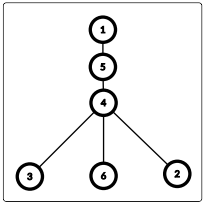


Figure 10.2: Hierarchical Collection

- Ordered in a structure similar to an upside-down tree
- Each node except the one at the top has just one predecessor (i.e. parent)
- Each node can have one or more successors (i.e. children)
- Node 4's predecessor (parent) is Node 5, and Node 4's successors (children) are Node 3, Node 6 and Node 2
- Example: A company's organizational tree

Ryerson University The Chang School of Continuing Education

Graph Collections

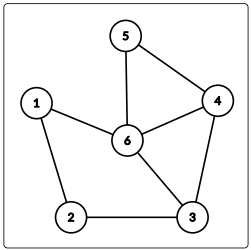


Figure 10.3: Graph Collection

- There is no limit to the number of predecessors or successors for each node
- All nodes connected to Node 6 are the predecessors and successors of that node
- Example: Maps of airline routes between cities

Unordered Collections

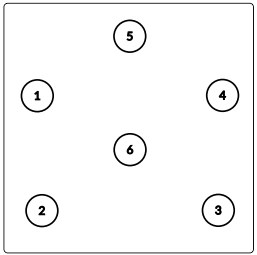


Figure 10.4: Unordered Collection

- No particular order
- Nodes are not connected, and each node does not have a specific predecessor or successor
- Example: A bowl of M&Ms

Operations on Collections

- Search and retrieval
- Removal, insertion, and replacement
- Traversal
- Test for equality
- Determine the size
- Cloning

Implementing Collections

1 Introduction to Collections

2 Implementation of Collections
Arrays Data Structures
Grids Data Structures

Arrays vs. Lists

- Similar to lists, arrays represent a sequence of elements that can be accessed or replaced at given positions
- Unlike lists, arrays are the primary implementing data structure in Python collections
- Arrays are more restrictive than lists, as arrays are created with a fixed length or size

Declaration of Arrays

```
1   class iArray():
2
3       def __init__(self, capacity, fillValue = None):
4           self._items = list()
5           for count in range(capacity):
6               self._items.append(fillValue)
7
8       def __getitem__(self, index):
9           return self._items[index]
10
11      def __setitem__(self, index, newItem):
12          self._items[index] = newItem
13
14      def __len__(self):
15          return len(self._items)
16
17      def __str__(self):
18          return str(self._items)
19
20      def __iter__(self):
21          return iter(self._items)
```

Efficiency of Arrays

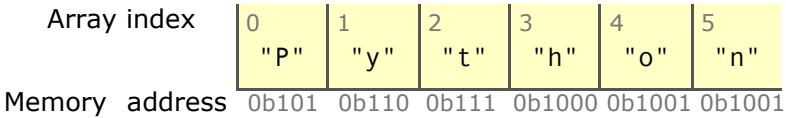


Figure 10.5: A fixed-size array with a base address of 0b101

- Indexing operation:
  1. Fetch the base address
  2. Return the required element's address

- Example: If the base address is 101_2, then the address of the data element at position 2 is ($101_2 + 010_2 = 111_2$)

Capacity of Arrays

- The capacity of arrays is determined when the arrays are declared
- When an array cannot hold more data, a larger one should be created, and all the elements should be transferred to the new one
- If the array seems to be wasting memory, it is preferable to decrease the capacity

Physical Size vs. Logical Size

0	1	2	3	4	5	6	7	8
"P"	"y"	"t"	"h"	"o"	"n"	None	None	None

Figure 10.6: A fixed-size array with a physical size of 9 and logical size of 6

- The **physical size** is the number used to specify the capacity when the array is created
- The **logical size** is the number of items inside the array
  - If the logical size is 0, then the array is empty
  - The index of the last item in the array is equal to the logical size minus 1

How Arrays Use Memory

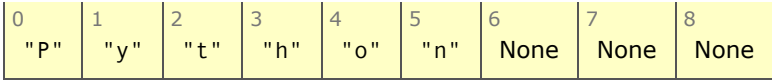


Figure 10.7: A fixed-size array with a load factor of $0.6\overline{6}$

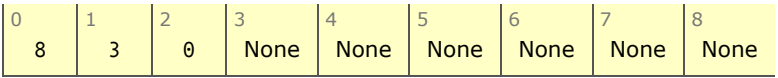


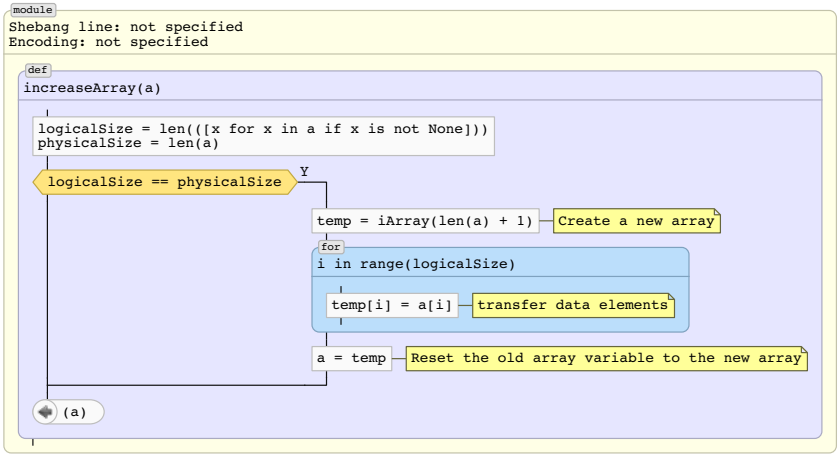
Figure 10.8: A fixed-size array with a load factor of $0.3\overline{3}$

The **load factor** is equal to the number of items stored in the array divided by the array's capacity

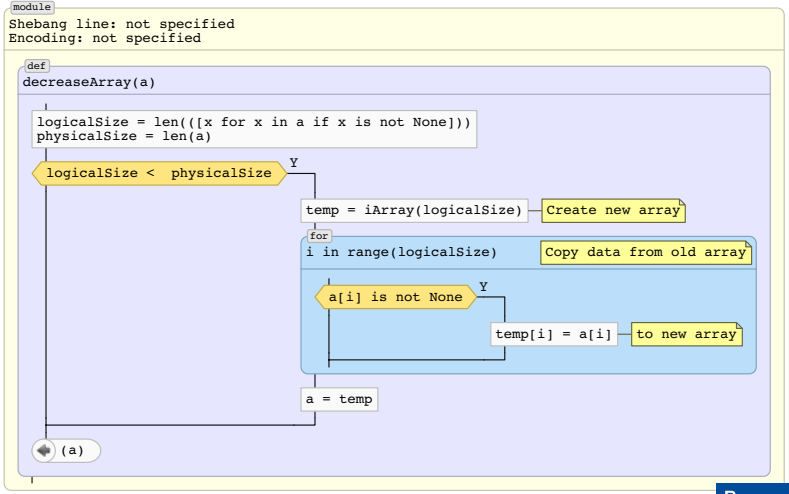
Operations of Arrays

- Increasing the size of an array
- Decreasing the size of an array
- Inserting an item into an array
- Removing an item from an array

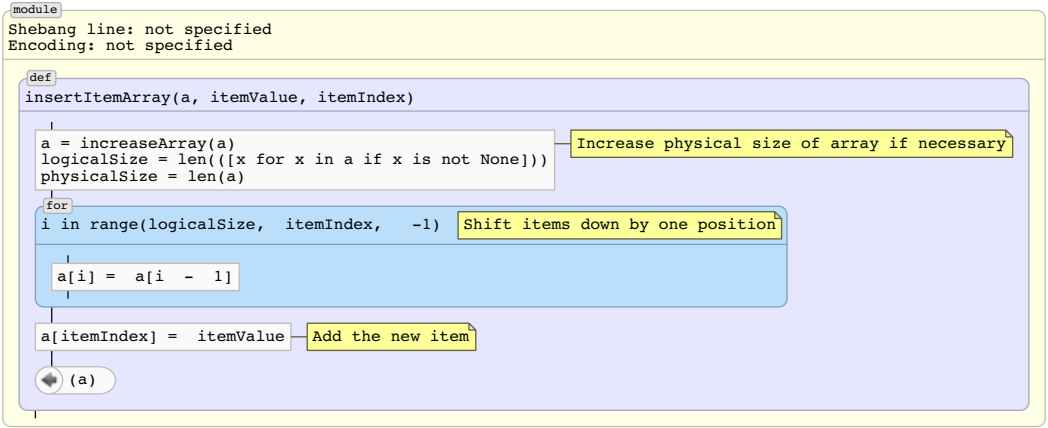
Increasing the Size of an Array (Control Flow)



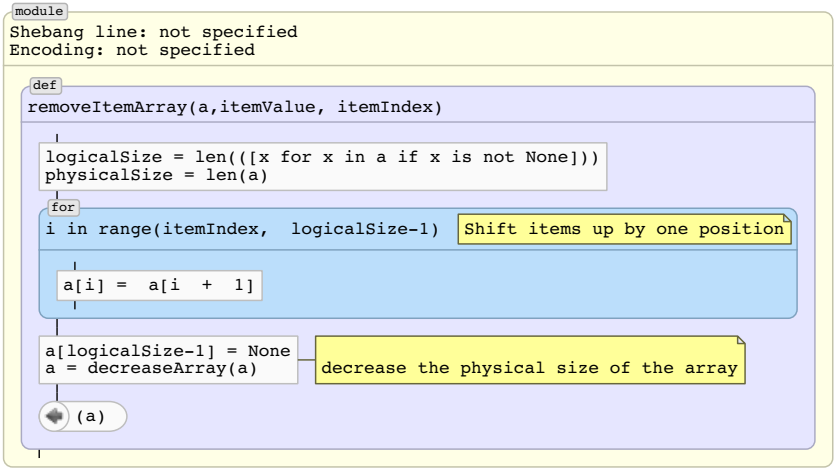
Decreasing the Size of an Array (Control Flow)



Inserting an Item into an Array (Control Flow)



Removing an Item from an Array (Control Flow)



Grids vs. Arrays

- A table of numbers can be implemented as a two-dimensional array or grid
- These grids are assumed to be rectangular and are of fixed dimensions
- To access an item in a grid, we use two subscripts to specify its row and column positions
  - Example: To access the value in row 2 and column 3, we use *grid*[1][2], as indexes start at 0

Declaring Grids

```
1   class iGrid():
2
3       def __init__(self, rows, columns, fillValue = None):
4           self._data = iArray(rows)
5           for row in range(rows):
6               self._data[row] = iArray(columns, fillValue)
7
8       def getRows(self):
9           return len(self._data)
10
11      def getColumns(self):
12          return len(self._data[0])
13
14      def __getitem__(self, index):
15          return self._data[index]
16
17      def __str__(self):
18          result = ' '
19          for row in range(self.getRows()):
20              for col in range(self.getColumns()):
21                  result += str(self._data[row][col]) + ' '
22              result += '\n'
23          return result
```

Processing Grids

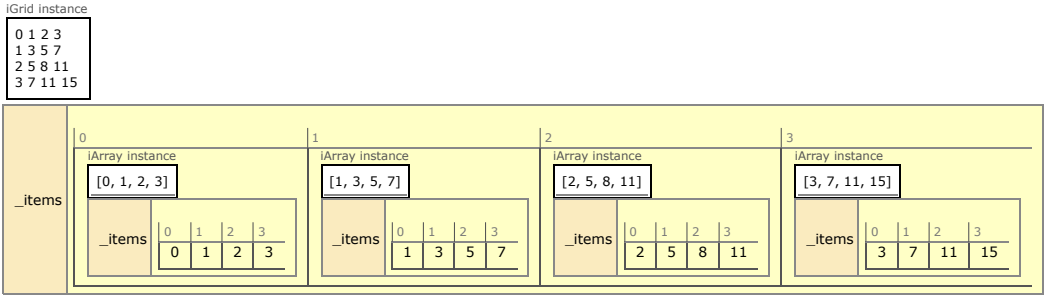
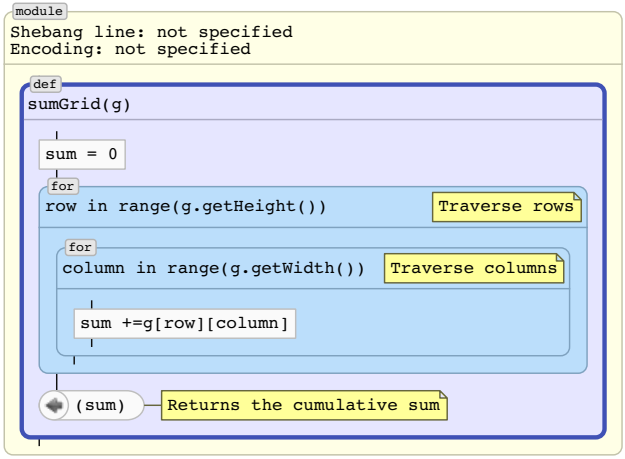


Figure 10.9: A fixed-size grid with 4 rows and 4 columns

Example 10.1 (Script)

```
1  def sumGrid(g):
2
3      sum = 0
4
5      for row in range(g.getHeight()):        #Traverse rows
6
7          for column in range(g.getWidth()): #Traverse columns
8              sum +=g[row][column]
9
10     return(sum)              #Returns the cumulative sum
```

Example 10.1 (Control Flow)



Ragged and N-dimensional Grids

- A cube of numbers can be implemented as a three-dimensional array or 3D grid
- To access a data element in a 3-dimensional grid, three subscripts are used to specify the item's row, column and outer-bound position
- For example, to access the value located in the 2^{nd} row, 3^{rd} column and 1^{st} outer-bound, we use *multigrid*[1][2][0], as indices start at 0
- Ragged grids have a fixed number of rows, but the number of columns in each row can vary

References

- Lambert, K. A. (2010). Collections, arrays, and linked structures. In *Fundamentals of Python: From first programs through data structures* (pp. 507-529). Boston, MA: Cengage Learning

- Python Software Foundation. (2020). Python 3.7.8 documentation. Retrieved from `https://docs.python.org/3.7/index.html`.